Creating a Safer OAuth User-Experience

Paul Youn — paul[at]isecpartners[dot]com

iSEC Partners, Inc
444 Spear Street, Suite 105
San Francisco, CA 94105
https://www.isecpartners.com

April 26, 2011

Abstract

An increasing number of web services are implementing OAuth servers in order to allow users to securely share their resources with third-party "consumer" applications. OAuth allows end-users to grant a consumer access to these private resources without surrendering their actual server credentials. Security risks can be introduced into an OAuth implementation and this paper suggests making a more secure user-experience by creating a simple and understandable workflow, implementing a least-privileges model, and auditing consumers.

1 Introduction

OAuth provides a framework that allows an end-user to authorize information sharing between web services or applications. The goal of OAuth is to allow resource sharing without requiring the user to share their actual credentials with multiple services. Although some background will be provided in this paper, it is recommended that the reader become familiar with the OAuth 1.0 protocol[1] as well as draft 15 of the OAuth 2.0 protocol[2][1]. Although the OAuth 2.0 is an evolution of OAuth 1.0, the overall security model is similar and most of the following recommendations apply to both versions.

1.1 The OAuth protocol

The OAuth protocol was designed as a three-legged authorization that involves an end-user, a consumer service requesting authorization, and a server hosting the end-user's resources[2]. The end-user initiates a request to share her resources across applications. The server is the party that controls the resources. The consumer is the party that will use the resources controlled by the server on behalf of the end-user.

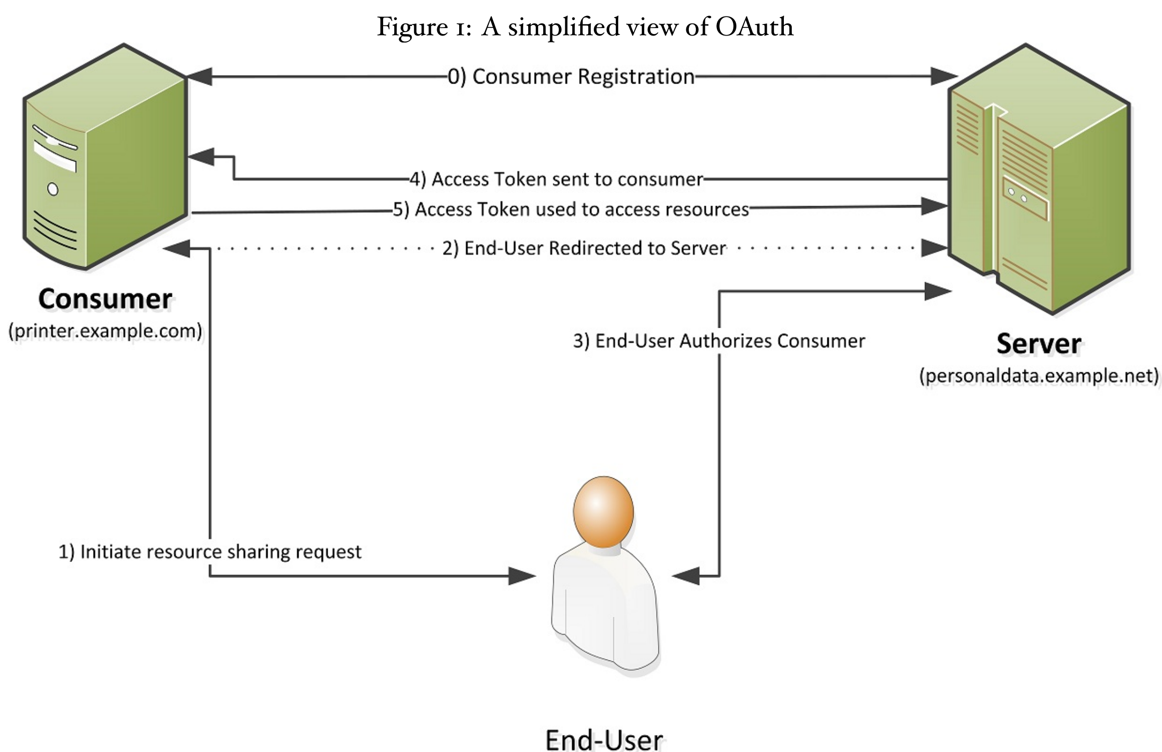
The example given in the OAuth 1.0 specification[1] involves a user Jane who wants to order prints from `printer.example.com` (the consumer) using pictures she keeps at another site `personaldata.example.net`[3] (the OAuth server). Jane wants to grant access of her photos to `printer.example.com` using the OAuth protocol.

[1] OAuth 2.0 may undergo significant changes before being finalized.
[2] Although OAuth 2 introduces several other use cases, the three-legged authorization scheme is the focus of this paper.
[3] The name of the website has been changed to highlight that the server may control more information than just photographs.

Figure 1 gives a simplified view of the OAuth protocol. In step zero, the consumer (`printer.example.com`) registers with the server (`personaldata.example.net`). Typically the server will issue a consumer ID and secret that the consumer uses to authenticate. Much of this paper focuses on using the registration process to implement least-privilege principles that protect a user from consumers that may ask the user for unnecessary privileges. A less discriminating registration process will treat a third-party that seeks to harvest data for advertising purposes the same as a third-party that responsibly handles the end-user's data. Although OAuth makes it tempting to place the responsibility on the end-users to protect themselves, this paper discusses several ways that the server and consumer can help the end-user by creating a safer environment.

In the first step of the typical authorization flow, an end-user initiates a request for the consumer (the printer service) to access private resources (Jane's photos) on the server. For the second step, the consumer (the printer service) packages an authorization request that contains a list of requested permissions, the duration of the credentials, and the redirect URI where the credential will be sent. The user is then redirected to the resource server.

Figure 1: A simplified view of OAuth



During the third step, the user must first authenticate to the server and is then asked to authorize the consumer's request. The authorization decision is all-or-nothing. A typical OAuth implementation will allow the consumer to request any permissions and rely on the user to decide if they want to use the service or not. The consumer may request permissions that are not required for the task at hand and the user will still grant the request if the service is valuable. For example, the printer service may request access to Jane's photos as well as access to her social media interface to broadcast that "Jane loves printing with `printer.example.com`!".

The authorized access token is sent to the consumer via the end-user in the fourth step. The consumer can now present the token to the resource server and gain access the end-user's resources in the fifth step[4].

[4]The interaction between server and consumer may require multiple round trips depending on the particular deployment scenario as described in the working draft of OAuth 2[2]

1.2 Beyond the OAuth specification

The OAuth specification only provides a protocol for exchanging security tokens. Consumer and resource provider sites wishing to use OAuth to collaborate still need to create a user-experience around authenticating users and authorizing access to resources. OAuth is intended to allow a user to easily share resources in a secure and controlled manner. Even vague authorization messages could expose end-users to risk because they won't be able to make informed decisions.

Implementors of OAuth should strive to protect the end-user and enable them to make informed decisions. A weaker implementation of OAuth may expose the user to consumers harvesting long-term and powerful credentials or insecure consumers that expose access tokens to attackers through bad practices. Understanding and mitigating the potential risks to the end-user is the subject of the remainder of this paper.

2 Towards a secure OAuth server framework

A complete implementation of the OAuth server must include an "authorization server"[5], a "resource server", and a means for managing consumers. All three of these major components will manage sensitive credentials and must be at least as secure as the server's normal authentication process. The authorization server must allow end-users to securely authenticate and grant access to their resources. The resource server must enforce access controls. The OAuth server must be able to register consumer applications, restrict or revoke consumer access, and audit consumer actions.

2.1 Follow the principle of least-privilege

Many consumers will only need very limited access to an end-user's resources in order to provide a service. For example, although an end-user stores a lot of private information on `personaldata.example.net`, `printer.example.com` only needs read access to certain photo albums. Although it is tempting to make the user responsible for granting least-privileges, the user is often faced with the choice of not using a consumer's service at all, or granting excessive permissions and accepting the risk. Even though the user has the final ability to grant or deny the authorization, an unrestricted consumer is able to request persistent excessive permissions as long as their service is valuable to the user.

A secure OAuth server should define resource sets that allow consumers to request and end- users to authorize the minimum access necessary for specific use-cases. A contact management system might need access to a user's address book or social graph, but not the full text of emails, their calendar, or their photos. A printing application might need read access to a particular photo album but not the address book. Although the permission scheme should be fine-grained, it must also be easy for the user to understand. Simplicity and clarity is preferable to technical correctness.

Once an OAuth server has implemented a permission scheme, consumers could specify exactly what permissions they would like to access at registration time. For example, the registration process should allow consumers to specify that they will only request permission to view photos and that tokens will only be valid for half an hour. The OAuth server can also curtail certain consumers based on reputation of the consumer, how securely the consumer secret will be protected[6], and what permissions are actually required to provide the consumer's service.

Unless the resource server itself has only one use-case, resource and permission sets should always be more granular than granting access to the full privileges of a user's account. This will not only help prevent abuse by

[5] Although this term was only coined in the OAuth 2 specification, OAuth 1 implementations often use an authorization server in practice.

[6] OAuth 2 also provides for use cases where clients do not use consumer secrets.

malicious or compromised consumer applications, but will make users more comfortable sharing access to their data. Resource providers that grant access to personal data unrelated to the tasks a user is trying to accomplish place their own reputation at risk if consumer applications are abusive.

Access tokens should be valid for a minimum amount of time for the task at hand. Even if a particular consumer requires long-lived tokens, all access tokens should be invalidated whenever the user changes their actual server password (or other credential). Failure to invalidate access tokens will create an awkward security model in which access tokens can be, in terms of longevity, more valuable than the user's original password.

2.2 Inform the end-user

Informing the end-user begins by authenticating the actual resource server. Just as with any web service interaction, the user must be confident that they are speaking with the server before authenticating and authorizing requests. Users must be trained to use browsers that display the OAuth authentication and authorization requests along with SSL indicators and the URL serving the request.

The authorization message should be simple and easy to understand. The message may also require internationalization, and real users should be consulted to see if the messages are being interpreted correctly. Live users should be able to provide feedback or request help if they are confused by an authorization message. By clearly informing the user, the server empowers the end-user to control their own private resources. If the OAuth server is not informative, the end-user may blame the server if an authorized consumer behaves unexpectedly.

2.2.1 User authentication

The OAuth server must authenticate the end-user before allowing them to authorize access to protected resources. Consider allowing the user to configure their account so they will be required to reenter their credentials even if they are already authenticated in order to prevent attacks such as cross-site request forgery.

Once the server has received a request from a consumer and authenticated an end-user for the first time, consider informing the user that a new consumer has requested access to their resources through an out-of-band channel such as email. This form of auditing will help end-users and OAuth servers to detect a security breach. See section 2.5 for a more thorough discussion of auditing.

2.2.2 The authorization message

The OAuth server should inform the end-user exactly what permissions are being requested, who is requesting the permissions[7], and how long the permissions will be valid (including if the token can be renewed without further authorization). The authorization message should be simple and possibly use intuitive icons with text representing resources.

Figure 2 is an example of an OAuth authorization screen[8] that provides poor information. Note that the identity of the calling application is included in the actual authorization message without any separation. Not only does this make it harder for the user to identify the originator of the authorization request, but if a malicious consumer could register an application name such as "yourself", the authorization message would become "Would you like to grant yourself access to your `vague.example.com` account?" In addition, the button labeled "Grant Access" may confuse a user into assuming they must grant access because closing the window is the only way to deny the request. Lastly, the message "The application will be able to access your account even if you are not logged in" doesn't indicate when the token will expire.

[7] The "oauth_callback" parameter (in OAuth 2.0, the "redirect_uri" parameter) specifies where the final authorization and access tokens are sent and is more relevant than the registered consumer name. The value of the "oauth_callback" parameter should be displayed to the

Figure 2: A less informative OAuth authorization screen

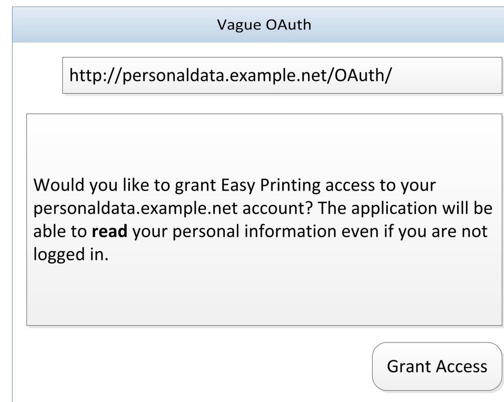
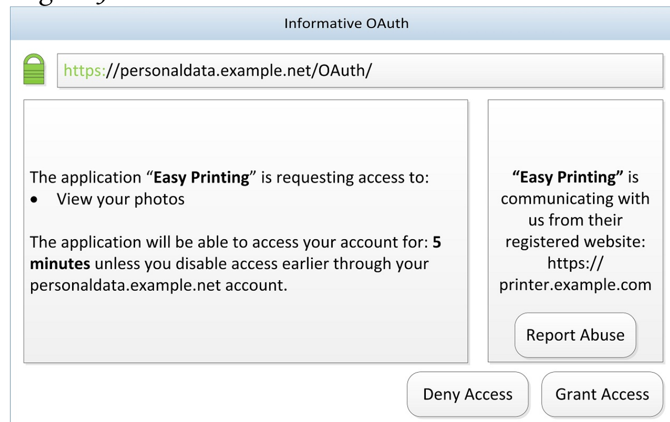


Figure 3 is an example of a more informative OAuth authorization screen. The identity of the calling application is clearly separated from the authorization message. A message is also displayed that could warn a user if the specified oauth_callback URI wasn't registered. The specific permission being requested is clearly presented. A message also indicates that the requesting application will have access to the data for five minutes. Lastly, the simple "Grant Access" or "Deny Access" options don't push the user toward one decision or another.

Figure 3: A more informative OAuth authorization screen



2.3 Consumer registration and management

Learn about a consumer before allowing them to register and request access to resources that are viewed as valuable. Not all consumers have the same security model and an OAuth server doesn't have to treat them equally. OAuth servers that only control low-value data may be willing to let any consumer application register and put the responsibility of authorization on the end-user, but this is generally not good enough for even moderately private or valuable data. Consumer application security flaws such as susceptibility to common flaws like Cross-Site-Scripting, SQL Injection, or poor web session management, will now reflect poorly on the server's service as a whole and put users at risk.

end-user in addition to the consumer name if a consumer did not register a callback URI.

[8]The example is based on actual implementations of OAuth authorization messages.

Protect against compromised consumer secrets. As suggested in section 2.1.1 of the OAuth 2[2] working draft, the server should require the consumer to register a valid callback URI whenever possible to make it harder for an impostor that has compromised a consumer secret to obtain access tokens. The server can then reject and flag any request for an access token to be sent to a non-registered URI. Secondary authentication mechanisms can also make it more difficult for a malicious attacker to spoof a consumer even if the consumer secret is compromised. Whenever possible, the callback URI should use SSL to authenticate and protect the access token sent to the consumer. The server should insist that all communications with the consumer and end-user are performed over SSL.

The OAuth server should clearly state security policies that govern how consumers can use the OAuth service and protect access tokens with explicit penalties for violations such as loss of service or other legal action. Requiring consumers to submit to random auditing can help enforce these policies.

2.4 Consumers that can not keep secrets

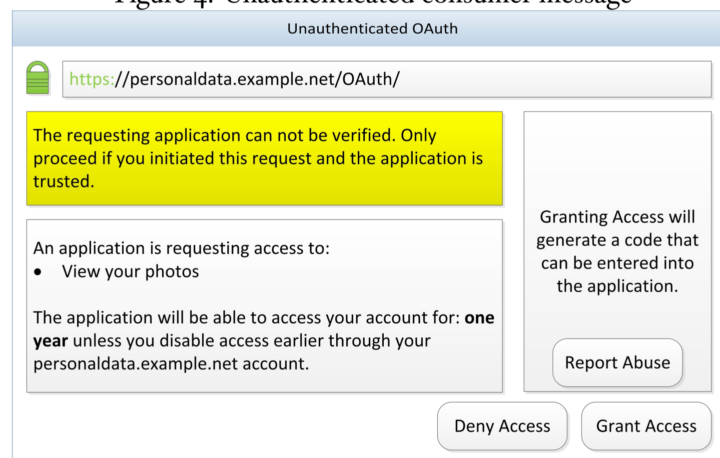
OAuth can still provide some value even if a distributed application is unable to protect a consumer secret. In some cases, it may be worthwhile for the server to support unauthenticated consumers but the decision should not be taken lightly and will allow an attacker to spoof popular applications. The following recommendations can provide some protections if unauthenticated consumers are supported.

Although a consumer secret should not be issued to applications that can not protect it, even unauthenticated applications may wish to persist a limited access token that can be independently managed by the end-user instead of actual credentials[9]. For defense-in-depth, the authorization server should consider curtailing the privileges that can be requested depending on how easy it is to positively identify a particular consumer.

Section 1.4.3 of the OAuth 2[2] working draft discusses a deployment scenario in which a consumer application receives and passes the user's actual credentials to the OAuth server in order to obtain a more limited OAuth access token. Although this deployment requires that the user trusts the consumer application, the OAuth server should honor the user's trust and issue an access token even if the consumer can't maintain a consumer secret. Because the consumer application already has the user's actual credentials, the added benefit of obtaining a lower privileged access token is minimal.

A consumer application that can't be authenticated may alternatively leverage the user's web-browser to obtain an OAuth token.

Figure 4: Unauthenticated consumer message



[9] The working draft of OAuth 2[2] highlights the use-case of a client that can't maintain secrets in section 3.

It is critical for the OAuth server to inform the user that the requesting consumer can't be authenticated rather than give the user a false sense of security by trusting the asserted consumer ID. Much like browsers support non-SSL traffic or allow a user to proceed even if an SSL certificate chain can not be verified[10], the key is to provide an OAuth experience that allows a user to understand the risks. Users have been trained to look for the SSL lock in browsers and similar mechanisms could be introduced for OAuth. Figure 4 shows a possible message that can be displayed to the user.

2.5 Audit and analyze consumer application requests

Inappropriate access to private data by even an authorized application can damage the reputation of an OAuth service provider, so it is important to audit consumers and monitor their activities for fraud and abuse. In some cases, a compromised or abusive consumer may be easy to detect: if the OAuth server receives a request using credentials registered to `printer.example.com` asking for access to write emails to the end-user's contact list, something is probably going wrong. Perhaps the request even specifies that the access token should be sent to a different URI than `printer.example.com` registered. The OAuth server should be able to automatically detect that the consumer is behaving unexpectedly and the request can be rejected and flagged. As mentioned in section 2.2.1, other types of abuse could be detected by notifying the user when a new relationship is established with a consumer to make sure the action was initiated by the user.

The OAuth server has two options for denying unexpected or malicious behavior. The server can block particular requests, or cut off all requests from a particular consumer. The second action is imprecise, and a protocol should be developed to work with the consumer to resolve the situation. Particularly valuable consumer-server relationships can be protected by issuing different consumer identities for different sets of requested permissions. Abuse of higher-privilege requests could be disabled without interrupting lower risk requests that are still behaving as expected until the issue is resolved.

Users should be engaged to assist with auditing because general auditing may be unable to detect certain abuses without an understanding what the end-user intended to authorize. To further protect against misbehaving consumers, the OAuth server should provide an interface that the end-user can use to invalidate existing access tokens. The interface should display all active access tokens as well as previously granted access tokens. Ideally, the end-user should be able to view a simple audit trail to see what resources have been accessed using each token. This audit trail wouldn't have to disrupt the user-experience, but a security conscious end-user could verify that their authorization token was used as expected and report abuse to the OAuth server owner. As with any logging or auditing scheme, the user's privacy should also be considered. Users should be educated about the auditing policy and be able to opt out of auditing that ties themselves to particular actions. Similarly, consumers should be informed of the server's auditing policy.

3 Consumers can reduce risk to the end-user

A secure consumer OAuth implementation will keep users happy and help maintain a strong partnership with the OAuth server. Security conscious servers and users may choose not to deal with consumers that don't protect an end-user even in extreme cases such as if a consumer secret is compromised or back-end data storage is exposed.

3.1 Only ask for only what you need

Self-policing will be very important for OAuth consumers. Because OAuth servers can deny a consumer access at any time for arbitrary reasons, it is in a consumer's best interest to maintain a trusting relationship with

[10] Typically after the user explicitly dismisses a warning.

the OAuth server. Consumers should specify what permissions they require during registration as well as valid "oauth_callback" parameter values whenever possible. If possible, a consumer should specify the lifetime for the requested access token. Consumers that ask for excessive permissions may damage their relationship with the server as well as the end-user. The requested permissions will be displayed to the user who won't be able to selectively grant permissions. If a consumer asks for too many permissions, the end-user may choose to deny the entire request.

3.2 Securely handle access tokens

The safest way to handle access tokens is to request short-lived or one-time-use tokens that have limited value to an attacker. If that isn't possible because of the consumer's requirements or limitations in the OAuth server, long-lived tokens should be deleted as soon as they are no longer needed. Remember that active access tokens are password equivalents for the authorized resources. If tokens need to be saved, a consumer must make best efforts to protect the token. The available protections will likely vary depending on deployment, but should be explicitly called out by a less secure consumer to explain the risks and mitigations servers will need to accept.

If the consumer application is a web server, ideally the token will only reside in memory, but it should be encrypted and protected with strict access controls if written to disc because tokens may have long-lifetimes that overlap with backup schedules or disc rotation. Avoid sending a token to a user's browser to reduce the risk of disclosure by common vulnerabilities such as Cross-Site Scripting. If tokens are sent to browsers, avoid placing them in cookies which can end up lasting longer than needed or being sent to places they don't belong. Also avoid placing tokens in URLs (i.e. as parameters in HTTP GET requests) as they may end up being exposed to other sites in referrer headings as well as logs.

If the consumer application is a distributed application used by more than one user at a time, it may be possible to secure user tokens in the cloud and off of the specific device. If a short-lived token is used, it can alternatively be stored in device memory, never written to disc, and cleared whenever it is no longer needed. Long-lived access tokens must be protected by distributed applications. The running device may contain a secret store that could be used to secure access tokens.

3.3 Secure large deployments

There are a couple of additional considerations for larger deployments of OAuth on top of all of the security concerns described above. High availability may be critical, and replication of OAuth secrets between redundant servers may be necessary. When replication is necessary an authenticated and encrypted channel should be used to transmit OAuth tokens.

Larger deployments may also have physically separated authorization servers and resource servers. When presented with an access token, the resource server must be able to verify that the token came from the authorization server. OAuth tokens may either need to be transmitted over an authenticated and encrypted channel between servers, or the token itself must be generated in a verifiable manner. For example, a shared secret could be used to create an HMAC of the token sent along as an additional optional parameter as allowed by the OAuth protocol.

4 Summary

It is the responsibility of an OAuth implementor to provide end-users with a secure and easily understood experience that allows them to make informed authorization decisions.

The OAuth server can create a more secure user-experience by clearly informing the user about exactly what resources will be shared, with whom, and for how long. The OAuth server should limit a consumer's privileges by providing a fine-grained privilege model that only allows consumers to request a minimum set of permissions. Regular reviews of the OAuth audit logs by security-conscious end-users and OAuth administrators will help hold consumers accountable if they don't comply with security policy.

The consumer should use the framework provided by the OAuth server to limit the risk if an OAuth access token is stolen or their consumer secret becomes known to an attacker. During registration, the consumer can protect themselves by specifying the permissions that will be requested, the lifetime of tokens, and valid callback URIs if possible. Once access tokens have been obtained, they must be protected. Long-term tokens should be avoided whenever possible, but strongly protected when necessary.

5 Acknowledgments

Thanks to Brad Hill and Jesse Burns for reviewing and suggesting significant improvements for this paper as well as other iSEC consultants Alban Diquet, Aaron Grattafiori, and Thomas Daniels for additional reviews.

References

[1] The oauth 1.0 protocol. http://tools.ietf.org/html/rfc5849, April 2010. 1

[2] The oauth 2.0 protocol, draft-ietf-oauth-v2-15. http://tools.ietf.org/html/draft-ietf-oauth-v2-15, April 2011. 1, 2, 6